COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

TR-963
AFOSR-77-3271

October, 1980

RECONFIGURABLE CELLULAR COMPUTERS

Azriel Rosenfeld
Angela Y. Wu*

Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, MD 20742

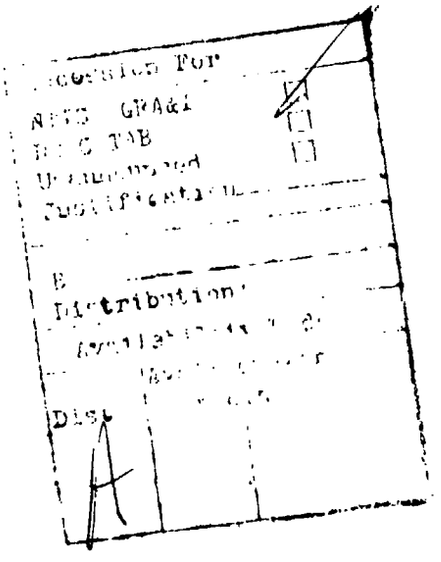
The support of the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3271 is gratefully acknowledged, as is the help of Sally Atkinson in preparing this paper. Some of the material in this paper is based on Technical Reports 730 (February 1979) and 790 (July 1979). The authors wish to thank Tsvi Dubitzki for his help in formulating some of the reconfiguration algorithms, and Todd Kushner for his help in computing expected graph diameters.

---

* Also with the Department of Mathematics, Statistics, and Computer Science, American University, Washington, DC.

ABSTRACT

When a collection of processors $C=\{P_1,\cdots,P_n\}$ operates in parallel, it is desirable that at any given stage of the computation, each P_i should have a task of about the same size to perform, and each P_i should require about the same amount of information from the other P's in order to perform its task. To the extent that these conditions are violated, parallelism is impaired, in the sense that the P's are not all used with equal efficiency. In cellular computers, e.g. as they might be used for parallel image processing, these conditions are maintained by having the P's all perform similar computations on different parts of the input data, and by allowing each P_i to receive information from a fixed set of the others (its "neighbors"), where these sets are all of bounded size. This paper discusses, on an abstract level, the concept of a reconfigurable cellular computer, in which each P_i can receive information from a set S_i of the other P's, and the S_i's are all of bounded size, but they need not remain fixed throughout a computation. Requiring the S_i's to have bounded size implies that most P's cannot communicate directly; the expected time required for two arbitrary P's to communicate depends on the graph structure defined by the sets S_i. The question of how to change the S_i's in parallel during the course of a computation is also discussed.



1. Parallelism and cellular computers

Let $C=\{P_1,\cdots,P_n\}$ be a collection of processors operating in parallel. In general, we can regard each P_i as performing a sequence of computational tasks, and at the end of each task, providing new information to other P's and requesting new information from other P's. In order to make efficient parallel use of the P's, we would like all of them to be active as much of the time as possible. This suggests that we should try to make the tasks as equal as possible in size, to avoid the need for some P_i having to wait a long time for a piece of information that some other P_i is still computing. Moreover, we should design the tasks so that each P_i needs to give about the same amount of information to other P's, and to receive about the same amount from other P's, between tasks, to avoid long I/O delays while some P_i is sending or receiving information.

Cellular computers [1-5] make efficient parallel use of large numbers of processors by dividing up both the computational taks and the I/O requirements very equally among two P's. On an abstract level, in a cellular automaton [6 -10], each computational step is symbolized by a change in a processor's state, and the new state depends on the old states of the processor and a fixed set of its neighbors; this corresponds to a task (lookup of the new state) that requires a fixed amount of new data (the neighbor's states) to be input (and by the same token, a fixed amount of data to be output: one's own

state to one's neighbors), and a fixed amount of computation. The same principles are used in concrete realizations of cellular computers, as applied to such tasks as image processing [1 - 5]. One can process an image using a square array of P's, each of which receives a block of image data, with neighboring P's receiving neighboring blocks. The processing is performed in stages, and at the end of each stage, neighboring P's exchange updated information about their blocks for use at the next stage. Thus here again, every P_i does essentially the same amount of processing and of I/O from/to neighboring P's, except that the amounts are somewhat less at the borders of the array. More general examples could be given in which the P's are connected to form a fixed graph structure (rather than an array structure), and are used to simulate interactions among the nodes of the graph (see [10]); note that here, too, we would want each node to have about the same number of neighbors, to keep their I/O requirements comparable.

It has usually been assumed, in studying cellular computers, that the number of "neighboring" P's with which a given P_i can communicate directly is bounded --i.e., if we represent the P's by the nodes of a graph, and join neighboring P's by arcs, the resulting graph has bounded degree, which does not grow with the number of P's. This assumption is very reasonable if we regard neighboring P's as hardwired together; the number of I/O ports available to a given P_i will be limited, no matter how many P's there are. But even if we do not assume hardwired connections,

it is still reasonable to require the number of neighbors of each P_i to be bounded, in order to put a bound on the amount of I/O that each P_i can do at a given stage of the computation. If we do not impose such a bound, different P's may require very different I/O delays, since some of them may need to output or receive much more information than others, so that once again there is danger of serious loss of parallelism.

In conventional cellular computers this graph structure defining the "neighbor" relations between P's is not only of bounded degree, but is also assumed to remain fixed in the course of a computation; this allows us to regard the neighboring P's as hardwired together. In this paper, we consider the possibility of reconfigurable cellular computers in which the set of neighbors of each P_i can change during the computation, but their number remains bounded. We do not consider here how direct communication is physically realized; we simply assume that each P_i has a list of "addresses" of those P_i's with which it can currently communicate directly, and that this list always remains of fixed size. (For the sake of concreteness [11], we can imagine that P_i communicates with P_j by putting a message addressed to P_j on a very fast bus.) We also assume that all communication is potentially two-way, i.e., if P_i can address P_j, then P_j can address P_i, and conversely.

When we assume, in a cellular computer, that the nodes are of bounded degree, we are making it easier to achieve efficient parallelism, but we are also introducing a potential speed

limitation due to the time that may now be required for information to be exchanged between two arbitrary P's. A given P_i can communicate directly only with a bounded subset of the P's, namely its neighbors, and if it needs to communicate with an arbitrary P_j, the message may have to be relayed through many stages. The expected and worst-case communication times between a pair of P's depend on the structure of the graph that defines the neighbor relationship; examples, for various standard graph structures, are given in Section 2. Evidently, cellular computers are best suited for tasks in which each P_i needs to communicate, for the most part, only with a bounded number of others, and their graph structures should be designed so that, to the extent possible, these others are P_i's neighbors.

In the case of a reconfigurable cellular computer, another problem arises when we want to change its graph structure during a computation. If P_i and P_j can currently address one another, it is easy for them to drop one another from their address lists by mutual agreement. But if P_i and P_k cannot currently address one another, how do they simultaneously add each other to their lists? Section 3 proposes a "local" approach to this problem, in which P_i and P_k can add each other to their lists only if they currently have a common neighbor P_j, which they may then simultaneously drop; and it is shown how, by iterating this "local reconfiguration" step, direct addressing can be established between any two

desired P's. In Section 4 we illustrate this approach by showing how various standard graph structures can be reconfigured, in parallel, into other standard structures, while maintaining boundedness of degree throughout.

2. Communication time in cellular computers

Let G be any undirected graph, with set of nodes N_G and set of arcs A_G. Two nodes P,Q are called neighbors if $(P,Q) \in A_G$. By a path of length m between two nodes P,Q we mean a sequence of nodes $P=Q_0,Q_1,\cdots,Q_m=Q$ such that Q_i is a neighbor of Q_{i-1}, $1 \leq i \leq m$. We say that G is connected if there is a path between any two nodes of G. We will usually assume in what follows that G is connected.

By the distance $\delta(P,Q)$ between P and Q we mean the shortest length of any path between them. [It is easily seen that distance is a metric, i.e. reflexive ($\delta(P,Q)=0$ iff $P=Q$), symmetric [$\delta(P,Q)=\delta(Q,P)$ for all P,Q), and satisfies the triangle inequality ($\delta(P,R) \leq \delta(P,Q)+\delta(Q,R)$ for all P,Q,R,).] The greatest distance between any two nodes of G is called the diameter of G, denoted $\Delta(G)$, and the expected distance between two randomly chosen nodes of G is called the expected diameter of G, denoted E(G).

Let C be a cellular computer with set of processors $\{P_1,\cdots,P_n\}$, and let A be the set of pairs of processors that (currently) can directly communicate with each other. If we let $N_G=\{P_1,\cdots,P_n\}$ and $A_G=A$, we obtain an undirected graph G, called the graph of C. The degree of a node P is the number of its neighbors, $d(P)=|\{Q|(P,Q) \in A_G\}|$. We say that G has degree d if $d(P) \leq d$ for all $P \in N_G$, where d is as

small as possible. We assume from now on that the graph of C always has degree$\leq$d for some fixed d.

The expected amount of time required for a message to get from one randomly chosen node to another is proportional to E(G), and the longest possible time for a message to get from one node to another is proportional to Δ(G). For a given number n of nodes, the values of E(G) and Δ(G) depend on the graph structure of G. Table 1 shows these values for a set of basic types of graphs. The derivations of the E(G) values are given in Appendix A.

Table 1 suggests that we can keep the expected or maximum communication time short by using high-dimensional trees or arrays as graph structures. However, such structures involve high node degrees, and the higher the degrees are, the more room there is for differences between the I/O requirements of different nodes. We will therefore consider only the low-degree cases from now on: string and cycle (degree$\leq$2), binary tree (degree$\leq$3), and two-dimensional array (degree$\leq$4).

Graph type	Maximum degree(d)	Diameter(Δ)	Expected diameter(E)
String	2	$n-1$	$(n+1)/3$
Cycle	2	$[n/2]$	$(n+1)/4$
Balanced binary tree	3	$2(\ell-1)$ where $\ell=\log_2(n+1)$	$\frac{2(n+1)}{n(n-1)}[(\ell-3)n+2\ell]$
Two-dimensional array	4	$2\sqrt{n}$	$2\sqrt{n}/3$
Balanced k-ary tree	k+1	2ℓ where $n=(k^{\ell+1}-1)/(k-1)$	---
k-dimensional array	2k	$k\sqrt[k]{n}$	---

Table 1. Values of diameter and expected diameter for some simple types of graphs, all having n nodes

3. Reconfiguration of cellular computers

Suppose that P_i and P_j can currently address one another, and P_i wants to drop P_j from its address list. Then P_j must drop P_i from its list simultaneously. To insure this, P_i sends P_j a message requesting that they drop each other; P_j acknowledges and agrees to the message; and they then drop each other. We assume here that such messages are sent and received in a unit time period, so that the dropping can take place simultaneously. Note that when two nodes drop each other, the graph may become disconnected; we will assume that normally this does not happen. (If desired, one can check that deletion of an arc will not disconnect the graph before actually deleting it; see [10].)

It is less obvious how P_i and P_j can add each other to their lists, if they cannot currently address one another. Suppose first that P_i and P_j have a common neighbor P_k. The sequence of events is then as follows: P_i (say) informs P_k that it wants to add P_j; P_k asks P_j to add P_i; P_j informs P_k that it agrees; P_k signals P_i to add P_j and P_j to add P_i simultaneously. Here again, standard unit times are assumed, to insure simultaneity. We have also assumed that P_i and P_j both have room to add each other without exceeding the degree bound. If this is not so, we can modify the construction to make P_i and P_j drop P_k at the same time they add each

other; this insures that their degrees remain within the bound. Of course, this assumes that there is no objection to disconnecting P_k from P_i and P_j.

In the case of an arbitrary P_i and P_j, we proceed by induction on the distance between them. (We assume the graph is connected, so that this distance always exists.) If the distance is 1, they are already neighbors; if it is 2, they have a common neighbor, and the construction in the previous paragraph can be used. Let the distance between them be $m>2$, and let $P_i=Q_0,Q_1,\cdots,Q_m=P_j$ be a shortest path between them. Then $P_i=Q_0$ and Q_2 have the common neighbor Q_1. By the previous paragraph, P_i and Q_2 can add each other to their lists and (if desired) can drop Q_1 from their lists. We now have a path $P_i=Q_0,Q_2,\cdots,Q_m=P_j$ of length m-1, so that the distance from P_i to P_j is now m-1. Repeating this construction, we can eventually reduce the distance to 2 and then to 1, at which point P_i and P_j have become neighbors. As before, we assume that there is no obstacle to adding and deleting the intermediate arcs involved in this construction.

The construction just given shows only how to create an arc between two arbitrary given nodes, and assumes that we are free to create and destroy intermediate arcs as needed. During the operation of a reconfigurable cellular

computer, many pairs of nodes may want to connect or disconnect themselves at the same time, and it will not in general be possible to carry out the necessary reconfiguration steps simultaneously without conflict. To demonstrate that the concept of reconfiguration is useful, we must show how graph structures can be nontrivially reconfigured in parallel. In Section 4 we will sketch several such reconfiguration algorithms which allow conflict-free parallel transformations between strings, cycles, arrays, and trees. We will generally assume in these algorithms, as is commonly assumed for graph-structured cellular automata, that the graph has a distinguished node.

4. Some parallel reconfiguration algorithms

4.1 Strings and cycles

It is trivial for a cycle to transform itself into a string by dropping an arc, e.g. one of the arcs incident on the distinguished node. Conversely, for a string to transform itself into a cycle, the node at one end (which we may assume to be distinguished) succesively connects itself to the third, fourth,...nodes, using the path-shortening construction in Section 3, until it is connected to the other end; each intermediate arc used in this construction is deleted as soon as the next arc is formed. The time required to form a string into a cycle is proportional to the length of the string.

4.2 String to tree

For a string, say of length ℓ, to transform itself into a balanced binary tree, a construction similar to that used for firing squad synchronization can be employed. The midpoint M_0 of the string (or one of the two midpoints, if ℓ is even) is identified by sending two signals from the (distinguished) end node, one at unit speed and one at 1/3 speed; the unit speed signal bounces back from the other end and meets the 1/3 signal at M_0. Next, we similarly find midpoints M_1 and M_2 of the two halves of the string, and at

the same time, we connect M_0 to each of them; M_0 is the root of the tree being constructed, and M_1, M_2 are its sons. We now have two substrings with midpoints M_1, M_2 and we repeat this process in parallel for each of them, thus joining M_1 to the midpoints M_{11}, M_{12}, of its halves, and M_2 to the midpoints M_{21}, M_{22} of its halves. After $\log_2 \ell$ repetitions of this procedure, we have constructed the tree. The total time required for the construction is about $\frac{3}{2}\ell + \frac{3}{2}\cdot\frac{\ell}{2} + \frac{3}{2}\cdot\frac{\ell}{2}2 + \ldots + \frac{3}{2} < 3\ell$, proportional to ℓ.

4.3 Tree to string

Given a binary tree, we can reconfigure it into a string in time proportional to the height of the tree. We do this by repeatedly, in parallel, joining each node to the right son of its left son and the left son of its right son, and disconnecting it from its left and right sons, where "left" and "right" refer to an arbitrary given labeling of the sons of each node. Figure 1 illustrates how this process works in a simple example. It is not hard to see that when the process terminates, each node is joined to the rightmost descendant of its left son and the leftmost descendant of its right son, and the resulting arcs define a string which corresponds to an inorder traversal of the tree.

4.4 String to array

We assume that each node knows the length ℓ of the string and its own position in the string. Let $s = \lceil \sqrt{\ell} \rceil$,

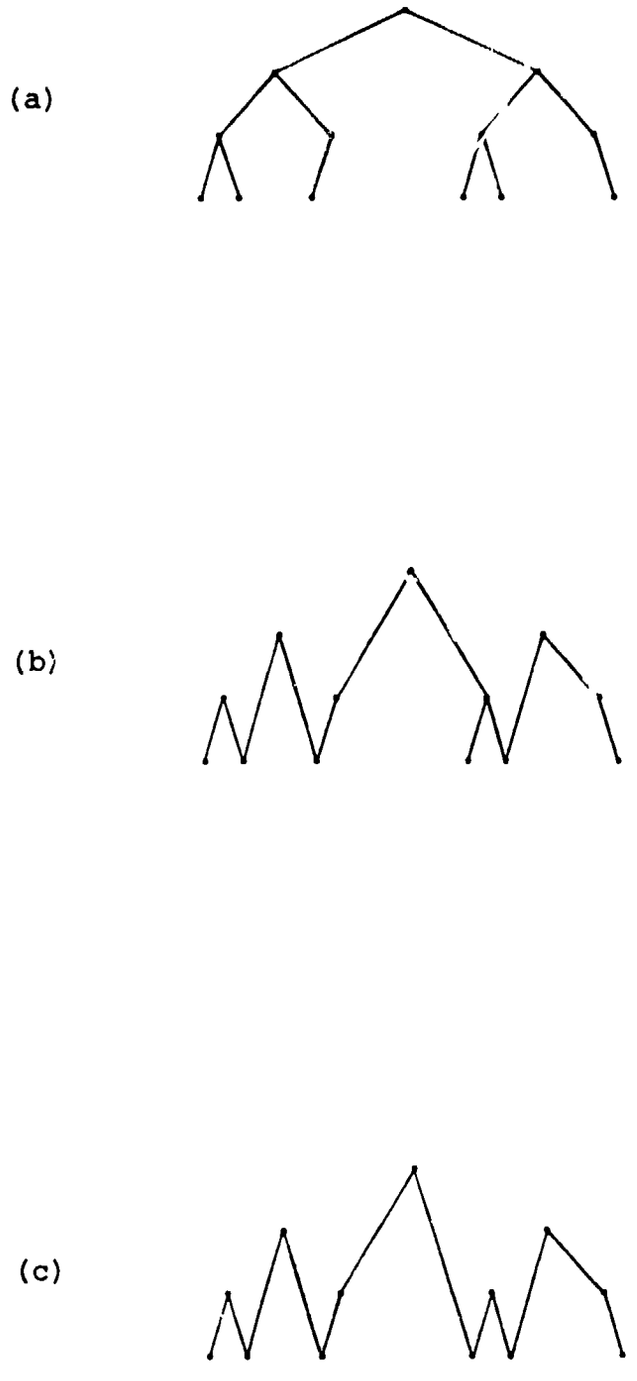


Figure 1. Reconfiguring a tree into a string

and regard the string as composed of substrings of length s. We join the i^{th} node of each substring to the i^{th} node of the following substring, $1 \leq i \leq s$, and disconnect the last node of each substring from the first node of the following one. Evidently, all these joinings can take place in parallel. [We do this as follows: Assume that nodes nos. 1, 1+2s.... are specially marked. Each marked node ks+i joins itself to the next marked node, using the stepwise construction of Section 3. As soon as this process has passed node ks+i, it too starts a reconstruction process, which stops as soon as it finds a node that still has only two neighbors and occurs after a marked node; this can only be node (k+1)s+i.] The substrings can be regarded as the rows of an array, and the new arcs connect the successive nodes in each column. If ℓ is not a perfect square, the last row will be shorter than the others. Evidently, the joining process takes time proportional to $\sqrt{\ell}$.

4.5 Array to tree or string

To change an array into a string, we can simply build a breadth-first spanning tree of the array with one of the corner nodes as root; readily, this tree is binary and can be constructed in such a way that it has height equal to the array's city block diameter. The construction of Section 4.3 can then be used to convert the tree into a string. The process takes time proportional to the array diameter.

For the details of a tree construction process that yields a tree of the desired height that is as balanced as possible, see Appendix B.

4.6 Tree to array

A binary tree can be converted into an array by first changing it into a string (Section 4.3) and then changing the string into an array (Section 4.4); but the latter process takes O(string length) time. A more complicated construction can be given which requires only O(array diameter) time; for the details, see Appendix B. It would be of interest to design an algorithm that requires only O(tree height) time.

5. Concluding remarks

This paper has suggested that it may be of interest to study reconfigurable cellular computers, in which the number of processors that can address a given one is bounded, but the set of these processors can change, thus modifying the graph structure defined by the addressability relation. Examples were given illustrating how various simple graph structures can be reconfigured into one another in parallel.

Ordinary cellular computers are applicable to computational tasks which can be divided among the processors in such a way that only certain pairs of processors need to interact; one would then define the graph structure of the computer so as to make these pairs neighbors. More generally, one could imagine a computational task in which, at various stages, different kinds of interprocessor interactions are needed. Such a task could be handled by a reconfigurable cellular computer which changed its graph structure at the end of each stage.

As an example of such a multistage task, let us again consider the domain of image processing. We know that an array-structured cellular computer is useful at an early stage of image analysis, when local operations are being performed on the image. The result of this stage might be a segmentation of the image into regions, and we might then want to perform further processing at the region level, e.g.

merging regions, or identifying particular configurations of regions by matching against models. This level of processing might be best carried out on a cellular computer configured in such a way that each node represents a region, and neighboring nodes represent adjacent regions. It is not difficult to define reconfiguration algorithms which, given an array-structured cellular processor in which region labels have been attached to the nodes, can construct a graph-structed cellular processor representing the adjacency graph of the regions. A paper describing such algorithms is in preparation [12-14].

Appendix A: Expected Diameters

1) Cycle

In a cycle of odd length n, the sum of the distances from any given node to the other nodes is

$$2\sum_{i=1}^{(n-1)/2} i = \frac{n^2-1}{4} .$$

Hence the average distance from an arbitrary node to any other node is the sum divided by n-1, or $(n+1)/4$. If n is even, the sum is

$$\frac{n}{2} + 2\sum_{i=1}^{(n-2)/2} i = \frac{n^2}{4}$$

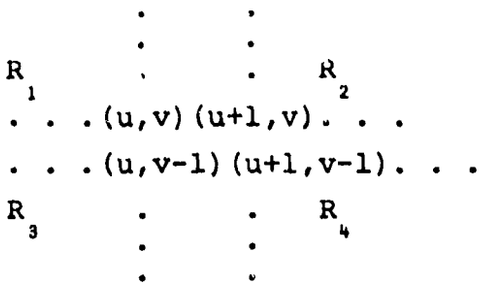
so that the average is $n^2/4(n-1)$. Note that if we include the given node itself (distance=0) in the average, the denominator is n rather than n-1, so that we obtain n/4 in the even case, and $(n^2-1)/4n$ in the odd case.

2) Array

In an r×s rectangle, the sum of the distances from any of the corner nodes to the other nodes is

$$\sum_{i=0}^{r-1}\sum_{j=0}^{s-1} (i+j) = \frac{rs(r-1)}{2} + \frac{rs(s-1)}{2} = rs(r+s-2)/2$$

Hence in a U×V rectangle, we can find the sum of the distances from a given node (u,v) to the other nodes by regarding (u,v) and three of its neighbors as the corners of four subrectangles:



Now a node in R_2 or R_3 has distance from (u,v) 1 greater than its distance from its own corner, while a node in R_4 has distance 2 greater. Hence the sum of the distances from (u,v) is the sum of the distances (+ 1 or 2) from the nodes of R_1, R_2, R_3, R_4 to their respective corners. Since the sizes of R_1, R_2, R_3, R_4 are $u\times v$, $(U-u)\times v$, $u\times(V-v)$, and $(U-u)\times(V-v)$, respectively, the sum is

$$\frac{uv(u+v-2)}{2} + (U-u)v\left[1+\frac{(U-u)+v-2}{2}\right]$$

$$+ u(V-v)\left[1+\frac{u+(V-v)-2}{2}\right] + (U-u)(V-v)\left[2+\frac{(U-u)+(V-v)-2}{2}\right]$$

which evaluates to

$$Uv^2+Vu^2-U(V+1)v-V(U+1)u + \tfrac{1}{2}UV(U+V+2)$$

and the average distance is this divided by UV-1. To obtain the average distance between a pair of arbitrary (distinct) nodes, we must average this result over (u,v), i.e. by taking $\frac{1}{UV}\sum_{u=1}^{U}\sum_{v=1}^{V}$ of it. Now applying this to v^2 yields $(V+1)(2V+1)/6$; to u^2, $(U+1)(2U+1)/6$; to v, $(V+1)/2$; and to u, $(U+1)/2$. Hence our final average is

$$\frac{1}{UV-1}\left[U(V+1)(2V+1)/6 + V(U+1)(2U+1)/6\right.$$

$$\left. - U(V+1)^2/2 - V(U+1)^2 + UV(U+V+2)/2\right]$$

which evaluates to $(U+V)/3$. In particular, for a square array of n nodes we have $U=V=\sqrt{n}$, so that the average is $2\sqrt{n}/3$; and for a string of n nodes we have U=n, V=1, so that the average is $(n+1)/3$.

3) Tree

A complete binary tree T of height h has $1,2,\ldots,2^{h-1}$ nodes at levels $0,1,\ldots,h-1$, respectively. For a node N at level k, the sum of the distances to the other nodes can be computed as follows: Let N be at distance d from the root of a subtree T' of height r which does not contain N. Thus N is at distance d+1 from 2 nodes of T', d+2 from 4 nodes, ..., and d+r-1 from 2^{r-1} nodes. The sum of the distances from N to the nodes of T' is thus

$$d + 2(d+1) + 2^2(d+2) + \ldots + 2^{r-1}(d+r-1)$$
$$= d(2^r-1) + \sum_{i=1}^{r-1} i2^i = d(2^r-1)+(r-2)2^r+2 = (d+r-2)2^r - (d-2).$$

Let N be at level k; then we can decompose T into subtrees as follows:

Root of T'	Distance^d from root of T' to N	Height^r of T'	Sum of distances from N to nodes of T'
N's brother	2	h-k	$(h-k)2^{h-k}$
N's father's brother	3	h-k+1	$(h-k+2)2^{h-k+1}-1$
N's grandfather's brother	4	h-k+2	$(h-k+4)2^{h-k+2}-2$
.	.	.	.
.	.	.	.
.	.	.	.
The brother of N's ancestor just below the root of T	k+1	h-1	$(h+k-2)2^{h-1} - (k-1)$

In fact, T consists of these subtrees together with N's father, grandfather, ..., and the root of T, which have distances $1,2,\ldots,k$ from N_1, hence sum of distances $k(k+1)/2$; and the subtree rooted at N itself, which has sum of distances $(h-k-2)2^{h-k} + 2$ from N. The contribution to the sum from the subtrees in the table is

$$(h-k)\sum_{i=1}^{k} 2^{h-i} + \sum_{i=1}^{k-1} (2i)2^{h-k+i} - \sum_{i=1}^{k-1} i$$

$$= (h-k)2^{h-k}(2^k-1) + 2^{h-k+1}((k-2)2^k+2) - k(k-1)/2$$

The total sum of distances for a node N at level k is thus

$$(h-k-2)2^{h-k} + 2 + (h-k)2^{h-k}(2^k-1) + 2^{h-k+1}((k-2)2^k+2) + k$$

$$= (k-2)2^{h+1} + (h-k)2^h + 2^{h-k+1} + k + 2$$

$$= 2^h(h+k-4)+2^{h-k+1} + k + 2$$

and the average distance is this divided by 2^h-2 (nodes $\neq$N).

To get the average distance between two arbitrary nodes, we take a weighted average of these sums, with weights 2^k (representing the 2^k nodes at level k, $k=0,1,\ldots,h-1$), and denominator 2^h-1 (the total number of nodes in T). This yields

$$\frac{1}{(2^h-1)(2^h-2)}\left[(2^h(h-4)+2)\sum_{k=0}^{h-1} 2^k + (2^h+1)\sum_{k=0}^{h-1} k2^k + \sum_{k=0}^{h-1} 2^{h+1}\right]$$

$$= \frac{1}{(2^h-1)(2^h-2)}\left[(2^h(h-4)+2)(2^h-1) + (2^h+1)((h-2)2^h+2) + h2^{h+1}\right]$$

For a tree having n nodes, we have $n=2^h-1$, so that this may be written as

$$\frac{2(n+1)}{n(n-1)}[(h-3)(n+1)+(h+3)] = \frac{2(n+1)}{n(n-1)}[(h-3)n + 2h]$$

where $h = \log_2(n+1)$.

Appendix B: Array/Tree and Tree/Array Reconfiguration*

In Section 4 we outlined a number of algorithms for parallel reconfiguration of one graph into another; but some of these algorithms were not the fastest possible or did not yield the best possible results (see Sections 4.5 and 4.6). The purpose of this appendix is to show how improvements can be achieved by using algorithms that are somewhat more complicated. Thus the appendix serves to illustrate that straightforward reconfiguration algorithms may not always be the best ones.

*The algorithms in this Appendix were developed with the help of Tsvi Dubitzki.

Algorithm B.1: Reconfiguring a two-dimensional array into a minimum-height binary tree.

Let A be a rectangular array of automata which contains N nodes where $N = r \cdot s$ ($r \leq s$) for integers r,s. D is the node at the northwest corner of A.

The basic steps of the algorithm are:

(1) Send a signal down from D along the leftmost vertical line. Upon receipt of this signal, each node below D along the vertical line sends a signal to erase the series of horizontal arcs emanating from it in A. This gives us an unbalanced binary tree with height at most r + s. We can view this tree as composed of one horizontal string of length s and s vertical strings of length r - 1. (The distinctions between left, right up and down connections at each node are known in A.)

(2) D sends a signal to order each string to turn into a balanced binary tree as described in Section 4.2. This takes at most O(s) time. We now have r + 1 binary trees: one with height $O(\lfloor \log s \rfloor)$ and s with height $O(\lfloor \log (r-1) \rfloor)$. In the above process the tree arcs are marked.

(3) Define the tree with s nodes as the "horizontal" tree T and the t trees with (r-1) nodes as "vertical" trees. We will hang the "vertical" trees on the leaves of the horizontal tree T. This is done as follows:

D sends a horizontal triggering signal through all the nodes of the tree T in A. Upon arrival at a node i (including D itself) the signal causes node i to check how many marked arcs of the tree are connected to it. If that number is 1 or 2 (except the root of T which is marked and considered as a node with 3 tree arcs) it means that respectively 2 or 1 of the "vertical" trees can be hung on node i in T. Then node i sends (ahead of the triggering signal) a searching signal for 2 or 1 roots of "vertical" trees either through the node below it in A or to the right, checking at each node whether the "vertical" tree below it, in A, is still connected to it. If it is still connected, then it can be assigned to node i of T, i.e. node i connects itself to the roots of its assigned trees and the arcs of A connecting these "vertical" trees to the upper horizontal line of A are disconnected. All the new connecting arcs to the roots of the "vertical" trees are marked as tree arcs. The horizontal triggering signal continues to the right one time unit after the searching signal starts, in order to avoid too many temporary connections at any node of T. In case the above searching signal, starting at node i, does not find enough needed unassigned "vertical" trees to its right, it bounces back to the left in the upper horizontal line of A to look for unassigned "vertical" trees left by the previous searching signals. This is not done when i is the rightmost node in A's top line.

(4) All the unmarked arcs (of A) are erased by a breadth first search signal from D sent down the spanning tree of A.

In the following a leaf is defined to be a node which does not have two sons in T and is said to have one or two null links.

Claim 2.1.1: There are enough null links at the leaves of T to hang all the "vertical" trees in A.

Proof: There are s nodes in T. By induction the number of null links in a binary tree with s nodes is s + 1. On the other hand there are only s "vertical" trees in A.

Corollary: If the rightmost node in A's top line finds under it one unassigned tree to be hung on it, then it doesn't bounce a signal back along A's top line since Claim 2.1.1 proves that there is one less "vertical" tree in A than needed to fill all the null links.

Claim 2.1.2: The height of the combined tree formed from T and the tree hanging from it is at most one unit more than the height of a balanced binary tree formed from a string of $N = s \cdot r$ nodes.

Proof: The height of a balanced binary tree with N nodes is $h = \lfloor \log_2 N \rfloor$. The total height of the combined tree constructed by Algorithm 2.1 will be

$$H = 1 + \lfloor \log_2 s \rfloor + \lfloor \log_2 (r-1) \rfloor \leq 1 + \lfloor \log_2 s \rfloor + \lfloor \log_2 r \rfloor \leq 1 + \lfloor \log_2 N \rfloor$$

so that $H \leq h + 1$.

Claim 2.1.3: Algorithm 2.1 takes O(s) time.

Proof: Step (1) of disconnecting the horizontal lines in A takes O(s+r) time.

Step (2) of converting all the strings into binary trees takes O(s) time.

Step (3) of converting the binary trees into one tree takes O(s) time.

Step (4) of erasing nontree arcs takes O(s+r) time.

Algorithm B.2: Reconfiguring a complete binary tree into a two-dimensional array.

Let T be a complete binary tree of automata with N nodes. Let D be the root of T. By a complete tree we mean a tree in which all the paths from the root to the leaves are of the same length. In the following a leaf node of T is a node with two null links.

The basic steps of the algorithm are:

(1) Conversion into a tree of strings:

In parallel D sends two signals down T, one at unit speed and the other at 1/3 speed. The unit speed signal bounces back from the leaves of T and meets the 1/3 speed signal at a node in the middle of each path from D to the leaves of T. Each such meeting node marks itself and turns the subtree rooted at it into a string as described in Section 4.3. The unit speed signals continue up to D and make it convert the binary tree rooted at it and having as leaves the marked nodes into a string also. We thus obtain a horizontal string (the last one) with two folded strings hanging from every other node of it, since in converting a binary tree into a string as described in Section 4.3, every two leaf nodes are separated by a nonleaf node, and the above twofold strings hang only on leaf nodes. D knows that the process of turning the specified subtrees into strings has terminated as soon as it receives (from its two sons in T) the string generating

signals which bounced back from T's leaves. All the arcs of T not participating in the above construction are erased as follows: D send breadth first erasing signals down in T. The signals bounce back from the leaves towards D and on their way back erase every arc of T except the first level of arcs above the leaves and above the marked nodes.

(2) Formation of a pseudo-array:

D orders every hanging point (in the horizontal string of (1)) of a twofold string to order the first node in the right part of the twofold string hanging from it to connect itself to the node to its right and then disconnect itself from its old hanging point. The rightmost node of the horizontal string doesn't have a nonleaf node to its right and therefore orders its right neighbor in the twofold string hanging from it to be a new hanging point to its right (thus part of the horizontal string) from which hangs the rest of the right part of that rightmost twofold string. We now have a binary tree composed of a set of strings hanging vertically from a horizontal string. This binary tree is a "pseudo-array" and we need only generate the horizontal connections in it in order to get an array. Note that the rightmost hanging string is one node shorter than the other hanging strings.

(3) Conversion into an array:

First we define for each node in the pseudo-array of step (2) what its upward, downward and horizontal connections are.

For this purpose D sends a breadth-first search signal down the pseudo-array. The signals bounce back from the bottom nodes of the vertical strings and go back up in the strings of the pseudo-array. Each entrance to a node in this path is a downward connection and each exit an upward connection. Upon arriving at the marked nodes of step (1) the definitions of the connections change to horizontal until the signals reach D again. Each node in the horizontal line will not emit a signal in the horizontal direction towards D until it has received a horizontal signal. Thus upon receiving two signals D will know that this marking process has terminated. At this stage D orders each of its horizontal neighbors to connect itself temporarily to the node on its downward connection. Then each of the horizontal neighbors of D orders its vertical neighbors and the node below D to connect themselves. The above temporary connections are then disconnected. In turn each horizontal neighbor of D starts such a connecting process too. This process propagates in the first upper row of the pseudo-array; at the same time each node below that row, having established a horizontal arc, starts such a process in the row below it, and so on until the network of horizontal arcs in D is completed.

Claim 2.2.1: The length of the string formed from the upper part of T (the upper row of the final array) is $O(\sqrt{N})$.

Proof: The number of nodes in T is N which equals $2^{h+1} - 1$ in a complete binary tree with height h. The marked nodes in step (1) of Algorithm 2.2 divide T into an upper complete tree with height h/2 and the rest of T. In that upper part of T we have $N' = 2^{h/2+1} - 1$ nodes. Therefore N' is $O(\sqrt{N})$.

Claim 2.2.2: Each hanging point in the pseudo-array of step (2) of Algorithm 2.2 is the middle of the twofold string hanging from it and the lengths of all the twofold strings in the pseudo-array are equal.

Proof: A complete binary tree has equal numbers of nodes in the right subtree and left subtree of its root. The subtrees forming the twofold strings in step (1) of Algorithm 2.2 are complete binary trees. The process of converting a binary tree into a string produces a string in which the root of the tree is an internal point, all the nodes to its right come from the right subtree of the root and all the nodes to its left come from its left subtree. Thus the root of the tree (a hanging point) is the middle of the twofold string. The lengths of all the twofold strings in the pseudo-array are equal since all the marked nodes of step (1) are at the same depth below D and hence all the subtrees below them are of the same size.

Corollary: The array formed in step (3) of Algorithm 2.2 is of size $O(\sqrt{N}) \times O(\sqrt{N})$. This is due to the fact that the upper horizontal line of the array contains $O(\sqrt{N})$ nodes by Claim 2.2.1

and the lengths of all the vertical strings hanging from the horizontal line of step (2) are equal by Claim 2.2.2.

Note that Algorithm 2.2 is applicable with slight changes to non-complete balanced binary trees. In particular if we are dealing with height-balanced binary trees with minimal numbers of nodes, then the upper horizontal line of the array holds less than $\sqrt{N}$ nodes since the marked nodes of step (1) (closest to the root) are now closer to D than in the case of a complete binary tree because of the existence of short paths going through a node to the leaves of T. Also the difference in length between the vertical hanging strings grows with N since we are dealing with subtrees (generating the twofold strings) which differ more and more in their numbers of nodes as the height of T grows. These factors give us finally very incomplete rectangular arrays.

Claim 2.2.3: Algorithm 2.2 takes $O(\sqrt{N})$ time.

Proof: Step (1) of constructing the tree of strings takes $O(\log N)$ time. Step (2) of constructing the pseudo-array takes constant time. Step (3) of forming the horizontal lines of the array takes $O(\sqrt{N})$ time since we already have a skeleton of an array of size $O(\sqrt{N}) \times O(\sqrt{N})$.

References

1. S. H. Unger, A computer oriented toward spatial problems, Proc. IRE 46, 1958, 1744-1750.

2. B.J. McCormick, The Illinois pattern recognition computer-ILLIAC III, IEEE Trans, EC-12, 1963, 791-813.

3. M.J.B. Duff and D.J. Watson, The cellular logic array processor, Computer J. 20. 1977, 68-72.

4. K.E. Batcher, Design of a massively parallel processor, IEEE Trans. C-29, 1980, 836-840.

5. P. Marks, Low level vision using an array processor, Computer Graphics Image Processing, 1980, in press.

6. A.R. Smith III, Cellular automata and formal languages, Proc. 11th SWAT, 1970, 216-224.

7. A.R. Smith III, Two-dimensional formal languages and pattern recognition by cellular automata, Proc. 12th SWAT,1971,144-152.

8. S.R. Kosaraju, On some open problems in the theory of cellular automata, IEEE Trans. C-23, 1974, 561-565.

9. A. Rosenfeld, Picture Languages, Academic Press, NY, 1979.

10. A. Wu and A. Rosenfeld, Cellular graph automata (I and II), Info. Control 42, 1979, 305-329, 330-353.

11. C. Rieger, ZMOB: A mob of 256 cooperative Z80A-based microcomputers, Proc. DARPA Image Understanding Workshop, November 1979, 25-30.

12. A. Wu and A. Rosenfeld, Local reconfiguration of networks of processors, TR-730, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, Maryland 20742, February 1979.

13. T. Dubitzki, A. Wu, and A. Rosenfeld, Local reconfiguration of networks of processors: arrays, trees, and graphs, TR-790, Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, Maryland 20742,July 1979.

14. A. Rosenfeld and A. Wu, Cellular computers for region-level image processing and analysis, in preparation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-81-0138	2. GOVT ACCESSION NO.: AD-A096005
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): RECONFIGURABLE CELLULAR COMPUTERS.	5. TYPE OF REPORT & PERIOD COVERED: Technical rept.
6. PERFORMING ORG. REPORT NUMBER: TR-963	
7. AUTHOR(s): Azriel Rosenfeld, Angela Y. Wu	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3271
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Math. & Info. Sciences, AFOSR/NM, Bolling AFB, Wash., DC 20332	12. REPORT DATE: October 1980
	13. NUMBER OF PAGES: 35
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 2304 A2	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

CSC-TR-963

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Cellular computers
Parallel processing
Reconfiguration
Cellular graph automata

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

When a collection of processors $C=\{P_1,\cdots,P_n\}$ operates in parallel it is desirable that at any given stage of the computation, each P_i should need to obtain about the same amount of information from the other P's in order to perform its task. To the extent that these conditions are violated, parallelism is impaired, in the sense that the P's are not all used with equal efficiency. In cellular computers, e.g. as they might be used for parallel image processing, these conditions are maintained by having

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403018

the P's all perform similar computations on different parts of the input data, and by allowing each P_i to receive information from a fixed set of the others (its "neighbors"), where these sets are all of bounded size. This paper discusses, on an abstract level, the concept of a reconfigurable cellular computer, in which each P_i can receive information from a set S_i of the other P's, and the S_i's are all of bounded size, but they need not remain fixed throughout a computation. Requiring the S_i's to have bounded size implies that most P's cannot communicate directly; the expected time required for two arbitrary P's to communicate depends on the graph structure defined by the sets S_i. The question of how to change the S_i's in parallel during the course of a computation is also discussed.